2015

Asynchronous Parallelization of a CFD Solver

Abdi, Daniel S.

Hindawi Publishing Corporation

Abdi, Daniel S., and Girma T. Bitsuamlak, "Asynchronous parallelization of a CFD solver," Journal of Computational Engineering, v. 2015 Article ID no. 295393, (2015), 10 p.

http://hdl.handle.net/10945/49514

Hindawi Publishing Corporation
Journal of Computational Engineering
Volume 2015, Article ID 295393, 10 pages
http://dx.doi.org/10.1155/2015/295393

Research Article

Asynchronous Parallelization of a CFD Solver

Daniel S. Abdi[1] and Girma T. Bitsuamlak[2]

[1]*Department of Applied Mathematics, Naval Postgraduate School, 1 University Circle, Monterey, CA 93943, USA*
[2]*Department of Civil and Environment Engineering, University of Western Ontario, 1151 Richmond Street, London, ON, Canada N6A 3K7*

Correspondence should be addressed to Daniel S. Abdi; dshawul@yahoo.com

Received 24 July 2015; Accepted 13 October 2015

Academic Editor: Fu-Yun Zhao

A Navier-Stokes equations solver is parallelized to run on a cluster of computers using the domain decomposition method. Two approaches of communication and computation are investigated, namely, synchronous and asynchronous methods. Asynchronous communication between subdomains is not commonly used in CFD codes; however, it has a potential to alleviate scaling bottlenecks incurred due to processors having to wait for each other at designated synchronization points. A common way to avoid this idle time is to overlap asynchronous communication with computation. For this to work, however, there must be something useful and independent a processor can do while waiting for messages to arrive. We investigate an alternative approach of computation, namely, conducting asynchronous iterations to improve local subdomain solution while communication is in progress. An in-house CFD code is parallelized using message passing interface (MPI), and scalability tests are conducted that suggest asynchronous iterations are a viable way of parallelizing CFD code.

1. Introduction

Atmospheric boundary layer simulations on complex terrains require tremendous amount of computational resources (CPU hours and memory) [1–3]. Even in the case of CFD simulation around a single building, tens of millions of grid cells may be required to fully resolve all scales of motion. The use of complex turbulence models, for instance, large eddy simulation instead of standard k-epsilon, further adds to the computational demand [4]. Therefore, one often resorts to methods that compromise accuracy for speed, such as use of wall models instead of resolving near wall flow [5], to get results within reasonable time. Parallel computation on a cluster of machines can help to get results faster without sacrificing quality of results. In this work, an in-house CFD program is developed and parallelized to run on a cluster of computers using two communication and computation approaches, namely, synchronous and asynchronous methods.

(1) Domain Decomposition. Wind flow simulations on complex terrain produce large amount of data at every time step of simulation, making it virtually impossible to simulate the whole domain using only a single commodity computer. To counter this problem, many CFD software programs use domain decomposition (DD) methods in which a processor takes care of only a part of the domain. Then, information is exchanged between the subdomains during the solution stage to enforce at least weak coupling. DD can be thought of as a "divide and conquer" strategy that can be used either when the problem is too big to fit in the memory space of one computer, or when the subdomains are more easily solvable than the original undecomposed problem. The method has been extensively used in aerospace engineering since early days to conduct finite element calculations on parts of an airplane [6]. In those days, computer memory was very limited; therefore, the only way to solve a big problem was to decompose it and solve each subdomain one by one by while imposing special boundary conditions to couple them. Some of the nonoverlapping DD methods are the Dirichlet-Neumann, Neumann-Neumann, and other adaptive variations suitable for hyperbolic convection problems. While the motivation for these methods was to solve large size problems that did not fit in the memory space of a single computer, the current

study is concerned with exploiting concurrency on a cluster of computers capable of holding the whole computational domain in distributed memory.

In this study, we implement an implicit parallel CFD solver and compare two approaches of communication and computation, namely, synchronous and asynchronous methods. The motivation for investigating the asynchronous method is that the synchronous method can incur significant performance loss on cluster of computers due to its use of collective communication calls, such as *MPI_Barrier*() and *MPI_Reduce*(). For example, calculating global error norms requires an expensive *MPI_Allreduce* call, that is often the subject of optimization on massively parallel computers. Gropp et al. [7] describe a way of parallelizing the Poisson equation using the synchronous approach and Jacobi iterations. The popularity of the synchronous method derives from the fact that computation done in parallel gives identical results, aside from floating point truncation errors, to one obtained with a serial computation. This gives great confidence to the developer and user that the parallel implementation is correct. However, synchronizing all processors at each step of iteration can be costly to scalability on thousands of processors. For instance, a recent study [8] on the open source CFD code OpenFOAM found barrier calls to be responsible for poor scaling performance. Synchronous communication is the standard way of implementing parallel CFD code and it has been used by [9–11] among many others.

The second alternative uses asynchronous communication and computation between subdomains to avoid all synchronization between processors. Use of asynchronous method in CFD is not common, but there are discussions and implementations by some researchers [12–14]. Some of them concluded that asynchronous computation has a better future because of heterogeneous clusters that have compute units with variable communication latencies, for example, CPU, GPU, and FGPA.

(2) Iterative Algorithms. Finite volume discretizations of the governing equations of fluid dynamics yield matrices that are highly sparse. The solution of such system of linear equations is carried out more efficiently using iterative algorithms, which successively update a solution vector starting from an initial guess, rather than using direct methods. Direct methods construct either the inverse of the matrix or its LU decomposition, which is both time consuming and also results in a full matrix even when the original matrix is sparse. Moreover, iterative methods allow for use of asynchronous iterations, which is the focus of the current study, to reduce idle time incurred due to the need for synchronization of processors. In the following, we will briefly discuss those iterative algorithms that are relevant to the current study.

The Krylov subspace methods, such as the preconditioned conjugate and biconjugate gradient methods (Algorithm 1), are popular among iterative algorithms due to their fast convergence properties. Even though simple relaxation algorithms (Algorithm 2) are not commonly used for solving linear system of equations all by themselves, they scale very well on large number of processors as mentioned in [15]. In addition, they can be used as preconditioners for the fast-converging Krylov subspace methods [16], and also as smoothers in a multigrid solver.

$r \leftarrow b - A * x_0$
$z_0 \leftarrow M^{-1} * r_0$
$p_0 \leftarrow z_0$
while not converged **do**
  $z \leftarrow A * p$
  $oo_r \leftarrow \mathrm{DOT}(p, z)$
  $\alpha \leftarrow o_r / oo_r$
  $x \leftarrow \mathrm{SAXPY}(x, p, \mathrm{alpha})$
  $r \leftarrow \mathrm{SAXPY}(r, z, -\mathrm{alpha})$
  $z \leftarrow M^{-1} * r$
  $oo_r \leftarrow o_r$
  $o_r \leftarrow \mathrm{DOT}(r, z)$
  $\beta \leftarrow o_r / oo_r$
  $p \leftarrow \mathrm{SAXPY}(p, z, \mathrm{beta})$
end while

Algorithm 1: Preconditioned CG.

Jacobi: $x^{(k+1)} = D^{-1}(b - (U + L)x^{(k)})$
GS: $x^{(k+1)} = (L + D)^{-1}(b - Ux^{(k)})$
SOR: $x^{(k+1)} = (1 - \omega)x^{(k)} + (\omega)x^{(k)}{}_{\mathrm{GS}}$

Algorithm 2: Relaxation.

The preconditioned conjugate gradient method (PCG) starts by calculating the residual r from an initial guess x_0 and then taking search directions p that will minimize the residual. PCG involves the following main operations: a matrix-vector multiplication, two dot products and three SAXPY vector additions ($y \leftarrow \alpha * x + y$), and matrix preconditioning operations. The convergence rate of PCG is highly dependent on the condition number of the matrix; thus, the preconditioner used is often more important than the solver itself [16]. The preconditioning is applied either through the direct inversion of the preconditioning matrix $z \leftarrow M^{-1} * r$ or more conveniently by solving an $M * z = r$ system of equations without inverting M. Diagonal preconditioners have advantages in this regard especially for the parallel version of the algorithm. Other preconditioners such as Incomplete LU factorization (ILU), symmetric successive overrelaxation (SSOR), and multigrid methods are harder to implement and also do not scale well on cluster of computers.

Given a decomposition of matrix A into its lower triangular (L), diagonal (D), and upper triangular (U) components as $A = L + D + U$, the most common relaxation methods, namely, Jacobi, Gauss-Seidel, and successive overrelaxation (SOR), can be formulated as updates of a solution vector $x^{(k+1)} = Gx^{(k)} + c$. The SOR method is an extension of Gauss-Seidel method that further accelerates convergence by overrelaxation. It combines newly computed values and old ones with a factor $\omega > 1$. SOR is convergent for $0 < \omega < 2$ for symmetric positive definite (SPD) matrices. A value of $\omega = 1.7$ gives good acceleration for many problems; however,

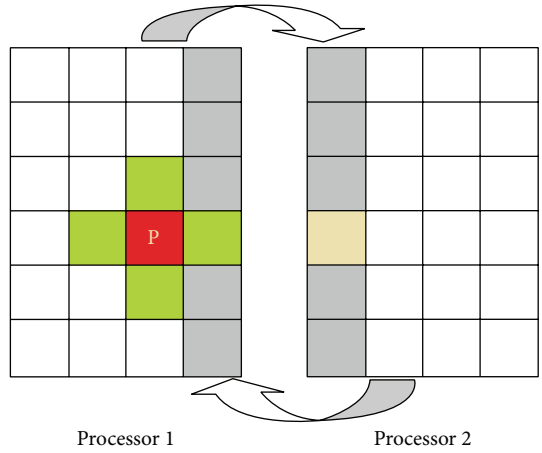


Figure 1: A 5-point boundary stencil for Jacobi iterations. The grey squares are halo layers used for keeping copies of values at off-processor cells.

larger values are commonly used since interest is usually in faster convergence rather than ensuring convergence. A symmetric version of the method (SSOR) does a forward sweep followed by another sweep in reversed order. SSOR converges slower than standard SOR with optimal ω value, but the motivation here is the symmetry of the iteration matrix which allows it to be used as a preconditioner for SPD matrices.

2. Synchronous Implementation

Iterative algorithms can be implemented in a strictly synchronized way so that the parallel version gives identical result as the sequential version at every iteration. Among relaxation methods, Jacobi iteration is the simplest to parallelize because new values (x^{k+1}) depend solely on old values (x^k). However, the method is not entirely embarrassingly parallel because some neighboring cells may reside in a different processor, thereby forcing communication between subdomains during solution. A 5-point stencil with an off-processor neighbor cell is shown in Figure 1. The size of the stencil depends on many factors: dimension of the problem, discretization method, the nature of differential equation being solved, and so forth. If the value of the neighbor on processor 2 is retrieved individually every time it is needed, parallel performance will suffer due to frequent small size message exchanges. This problem can be solved by adding boundary cells, also known as *halo* layer, around shared interface to keep local copies of off-processor neighbors. The values stored in halo layers are exchanged at the end of each iteration. In addition, updates of boundary stencils follow same procedure as that of internal stencils without needing to check if a neighboring cell is off-processor or not.

Wide halo layers (two or more halos) may be used when the stencil encompasses neighbors two or more steps away from the center cell. Besides increasing coupling between subdomains, wide halos also help to reduce communication overhead because inner halo layer can be locally updated without the need to exchange data at every iteration. To be more precise, communication needs to be done only once every nth iteration for an n-halo layer approach. Information exchange can be done through point-to-point communication calls, MPI_send, and corresponding *MPI_irecv* calls in every processor. A strict order of message exchange should be enforced to avoid deadlock situations where two processors are simultaneously waiting for messages from each other. This blocked communication scheme also adds synchronization points, albeit locally between two neighboring processors, that are easily avoidable. Both problems can be tackled by an asynchronous communication scheme via MPI_isend and *MPI_irecv* calls as outlined in Algorithm 3. The synchronization is done once with one MPI_Waitall call at the end unlike the synchronous case.

The most important aspect of the asynchronous communication scheme is that it allows for computation-communication overlap. Computation on internal stencils can be done while the halo layer values are being communicated at the boundaries. Algorithm 4 shows an outline of this process. First, communication of inter-processor boundary values is initiated with asynchronous sends, and then calculations at interior cells, which do not require halo layer values, are conducted. Calculations at boundary stencils are done only after communication is completed, that is, after the *MPI_Waitall*() call.

Parallelizing Gauss-Seidel and SOR algorithms is not as straight forward as parallelizing Jacobi. This is because the methods use values from the current iteration as soon as they become available bringing in sequential dependence. Depending on the order of the original equations, different results can be obtained at each iteration leading to basically different Gauss-Seidel methods. This is problematic for validation of parallelly computed results against serially computed results. However, Gauss-Seidel has superior convergence properties than Jacobi and is known to converge twice as fast asymptotically. To counter this problem of parallelization, graph coloring algorithms can be used to break down a single sweep of SOR into two or more equivalent sweeps that can be applied in parallel. A red-black

```
procedure EXCHANGE
    for all to ← neighbors do
        MPI_isend(to)
        MPI_irecv(to)
    end for
    MPI_Waitall()
end procedure
```

ALGORITHM 3: Asynchronous communication.

```
procedure EXCHANGE-AND-COMPUTE
    for all to ← neighbors do
        MPI_isend(to)
        MPI_irecv(to)
    end for
    Jacobi update for internal stencils
    MPI_Waitall()
    Jacobi update for boundary stencils
end procedure
```

ALGORITHM 4: Asynchronous communication with computation-communication overlap.

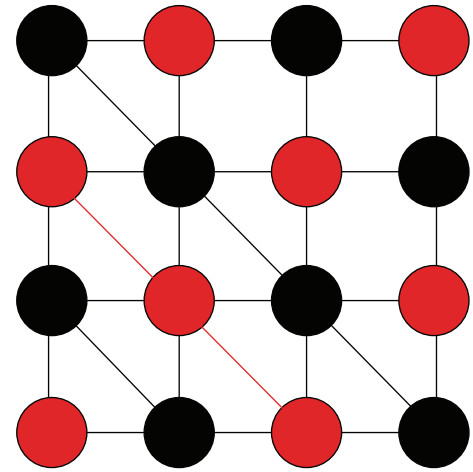


FIGURE 2: Red-black coloring of elements used for parallel Gauss-Seidel. Black elements have all red elements and vice versa; hence, updates on each color can be done in parallel.

coloring of elements is shown in Figure 2. Another method known as wavefront exploits parallelizability of computations on diagonals of matrices. A source of concern with these methods is load balancing. The coloring algorithm should ensure that approximately equal amount sof nodes are assigned to each color on all processors; otherwise, the idle time spent waiting for processors with larger portion of work becomes a bottleneck. The wavefront method is destined to have unequal amount of work at different diagonals; thus, it inherently suffers from this problem. An advantage of wavefront method over graph coloring is that it preserves the original order, and thus it has the same convergence rate as its sequential counterpart. It is known that reordered Gauss-Seidel converges slower than the sequential counterpart that has a natural ordering. The first sweep in a red-black Gauss-Seidel is basically a Jacobi iteration since no values from the current iteration are used. Thus, the overall red-black algorithm will have convergence rate equivalent to Jacobi-GS sweeps. The wavefront method uses values from the current iteration, but it offers significantly less parallelization than graph coloring algorithms.

Asynchronous Gauss-Seidel (chaotic relaxation), pioneered by Chazan and Miranker [17], is an easier alternative that can avoid the above complications if convergence can be reached. Besides ease of implementation, asynchronous method does not need to exchange halo layer values at every iteration thereby completely avoiding the associated latency. This is especially so in the case of irregular regions and irregular partitions or in inhomogeneous clusters [18, 19]. Synchronization is avoided at all stages of solution; however, the method may take larger number of iterations to converge or sometimes not converge at all. Halo layers are updated as the neighbor processor sends them; therefore, it is difficult to analyze convergence property of the method. However, the conditions under which convergence is achieved have been studied by Baudet [20].

Parallelization of preconditioned conjugate gradient (PCG) solver involves different stages that impact performance differently. These stages are outlined in Algorithm 5. The scalar operations SAXPY are embarrassingly parallel with no communication required whatsoever; thus, this operation scales well. The operations that are bottlenecks of scaling are the matrix-vector product, reduction, and preconditioning stage. The EXCHANGE(p) operation at the beginning makes sure that halo layers have the latest values before local matrix-vector multiplications are done. The operation has an implicit barrier at the end that further adds to synchronization overheads. However, this can be avoided by overlapping computation with asynchronous communication, just like what is done for Jacobi in the previous section. Internal updates in the matrix-vector multiplication, which do not require values from adjacent processors, can be done while halo layer values are being communicated asynchronously [15, 21]. The local DOT products can be done in parallel; however, the ensuing summation of local products, that is, REDUCE operation, introduces many synchronization points. This operation is commonly done through smart algorithms that are able to do the calculation in $O(\log_2(N))$ time. Unlike the EXCHANGE(p) operation that involves only adjacent processor, REDUCE forces all processors to synchronize; thus, its impact on performance is high. The matrix preconditioning stage is difficult to parallelize except when the preconditioner matrix is of diagonal form, for example, Jacobi or diagonal ILU preconditioner. A strict synchronous implementation of PCG also requires calculation of global residuals via collective communications such as MPI_Allreduce. Due to complexity of implementing a parallel preconditioner that gives identical result as its sequential counterpart, domain decomposition method with

EXCHANGE(p)
$z \leftarrow A * p$
$oo_r \leftarrow \text{DOT}(p, z)$
$oo_r \leftarrow \text{REDUCE}(oo_r)$
$\alpha \leftarrow o_r / oo_r$
$x \leftarrow \text{SAXPY}(x, p, \text{alpha})$
$r \leftarrow \text{SAXPY}(r, z, -\text{alpha})$
$z \leftarrow M^{-1} * r$
$oo_r \leftarrow o_r$
$o_r \leftarrow \text{DOT}(r, z)$
$o_r \leftarrow \text{REDUCE}(\text{type}, o_r)$
$\beta \leftarrow o_r / oo_r$
$p \leftarrow \text{SAXPY}(p, z, \text{beta})$

Algorithm 5: Parallel PCG.

local matrix preconditioning is often used. This will result in more iterations but maybe worth the cost in light of its advantage for scalability on many processors. This is also the approach followed for parallelizing the current program.

3. Asynchronous Implementation

In the previous section, methods of implementing parallel algorithms that strictly follow same computational path as their sequential counterparts, and hence give identical results, have been discussed. The work required for a strict implementation can sometimes be overwhelming. A significant reduction in complexity can be achieved by relaxing some requirements. For example, use of chaotic relaxation avoids the need for complex algorithms such as graph coloring and wavefront method that are required only for enforcing the parallel implementations to yield identical results as the serial version. Also using local preconditioning on subdomains, hard-to-implement preconditioners such as parallel ILU, can be avoided. The number of synchronization points introduced for parallelizing PCG solver also suggests potential scalability issues on massively parallel systems. Given the above difficulties, it is worthwhile to investigate asynchronous algorithms. In these methods, each processor does its own calculations without waiting for other processors to finish part of their work. As long as halo layers are updated regularly, one processor could be solving fluid equations while the other solves solid equations; one processor could be using PCG and the other SOR, and so forth. This complete freedom comes at a price of increased number of iterations or in some cases divergence of solution, and nondeterministic results as well. However, its advantage regarding scalability can be a deciding factor with the everincreasing computational power with thousands of processors, and load balancing problems on inhomogeneous clusters.

An asynchronous implementation of solvers is outlined in Figure 3. All the iterative algorithms discussed so far can benefit from asynchronous iterations provided that convergence can be achieved. Unlike the case of synchronous implementation, information does not need to be exchanged at the end of each iteration. Also for asynchronous PCG, processors do not need to synchronize after calculating local DOT product (REDUCE operation), but instead they use the local values to calculate α and β. Each processor probes for messages from its neighbors using MPI_iprobe at the end of the iteration, and if there is one the halo layers are updated with new values. Immediately afterwards, the processor sends back either its own halo layer data at the shared boundary or an END message to signal convergence has been reached on its local problem. A processor must reply with a message whenever it receives one to keep the cycle of communication going. Each processor also keeps count of how many of its neighbors reached convergence and then stops calculations when all its neighbors and itself reach convergence. Whenever a message is received, the residual and search directions for PCG are recalculated to basically restart the PCG iterations by incorporating the effect of new halo layer values. This can sometimes lead to more number of iterations than would be necessary for a synchronous implementation. While the convergence properties of asynchronous iterations for Gauss Seidel are established [20], that for Krylov methods is not well known. The experience from our implementation seems to work most of the time, but sometimes it can lead to many iterations probably due to frequent restarts of the algorithm.

4. Validation with Lid-Driven Cavity Flow

The well-known lid-driven cavity benchmark problem [22, 23] is used to validate the asynchronous parallel implementation. The problem concerns a two-dimensional fluid flow inside a closed container driven by a moving lid at the top as shown in Figure 4. Dirichlet boundary conditions are applied around the unit square container: $u = 1$, $v = 0$ at the top and no-slip $u = v = 0$ in all others.

The streamline plots for this 2D flow case at different Reynolds numbers using a uniform grid of 128×128 are shown in Figure 5. The results from our implementation are validated against the spectral analysis of the lid-driven cavity flow by Botella and Peyret [22]. Plots of u on vertical section and v on horizontal section for the Re = 1000 case show excellent agreement as shown in Figure 6. The streamlines for higher Reynolds number are also compared with plots found in Goyon [23]. The flow structures for the primary and secondary vortices show very good similarity.

Next, we solve a 3D version of the lid-driven cavity problem on a unit cube by decomposing it into 16 subdomains as shown in Figure 7. The solution at all interprocessor boundaries does not show jumps, which indicate that the iterative solutions in each subdomain have reached full convergence. We have also tested our parallel implementation on more complex problems with unstructured mesh and using graph partitioning software METIS to decompose the domain. In all cases, the results found using the asynchronous implementation are in agreement with that of synchronous implementation. As discussed in previous sections, asynchronous algorithm may sometimes diverge where a synchronous algorithm would not, and this has been observed in some of our tests.

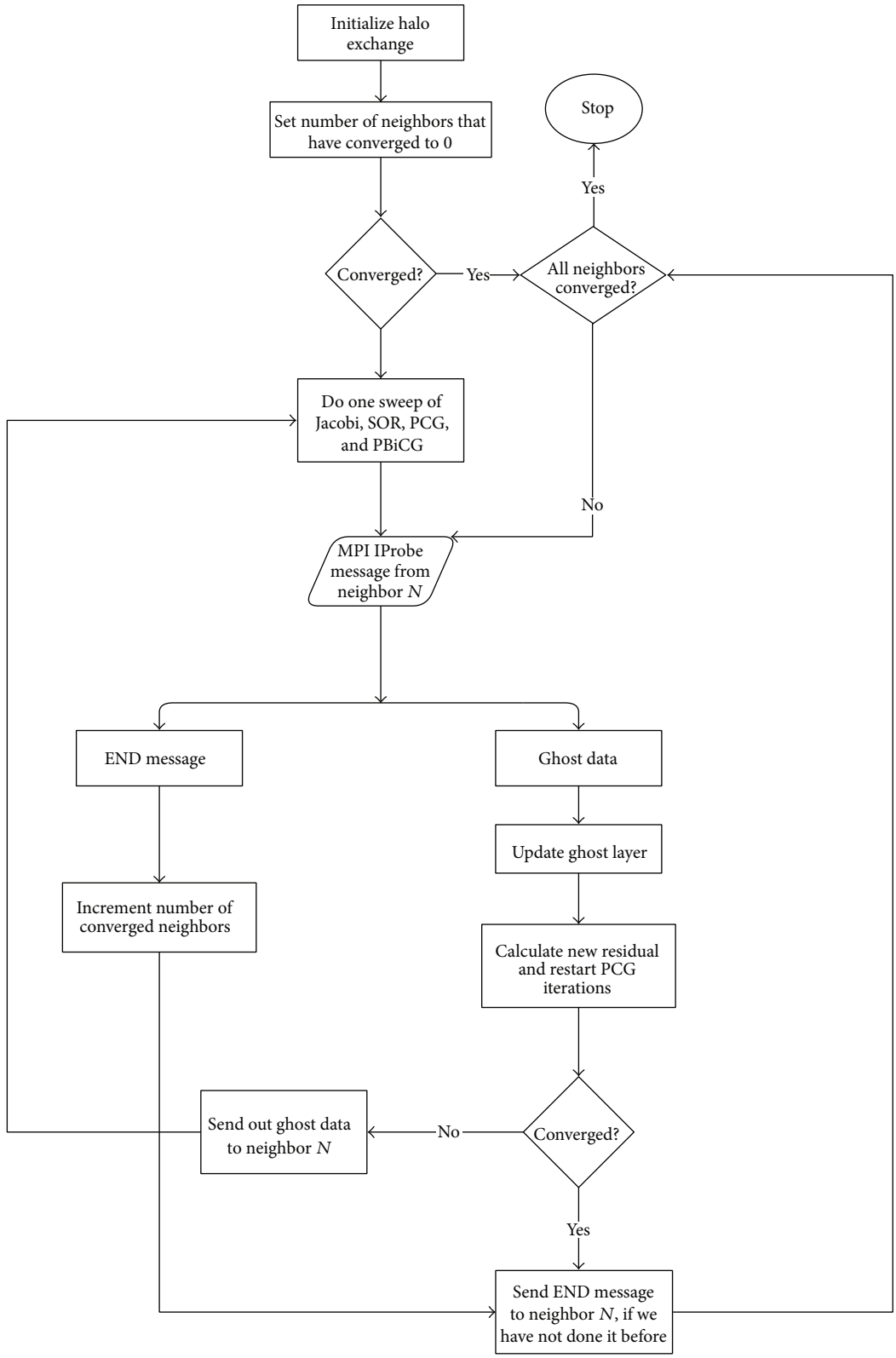


Figure 3: Asynchronous solution: messages are processed as they arrive *asynchronously*. Each worker checks for messages from its neighbors after conducting one iteration of the solver. For every message a processor receives, it must send back its own message to keep the cycle of message exchange going. If no message is received, the processor does *asynchronous iterations* to improve its local solution.

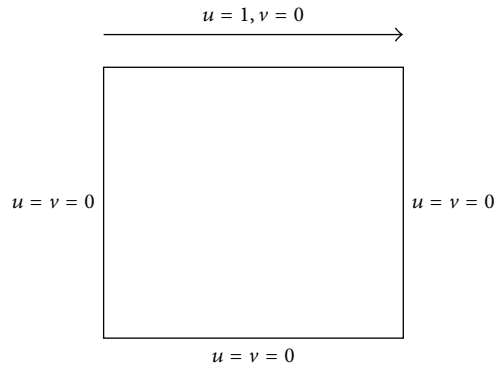


Figure 4: Lid-driven cavity flow: the flow is driven by the sliding lid at the top. A no-slip boundary condition is applied at all other sides of the square container.

Table 1: Internode scalability for 256 × 256 case.

Processors	Time (ms)	Speedup
1	1427135	1.00
2	772862	1.85
4	427012	3.34
9	198425	7.19
16	124985	11.42
25	92477	15.43
36	84422	16.90

Table 2: Internode scalability for 1024 × 1024 case.

Processors	Time (ms)	Speedup/16	Speedup/1
16	4100112	1	11.42
25	2819571	1.45	16.5
36	2037217	2.01	22.84

5. Scalability Study

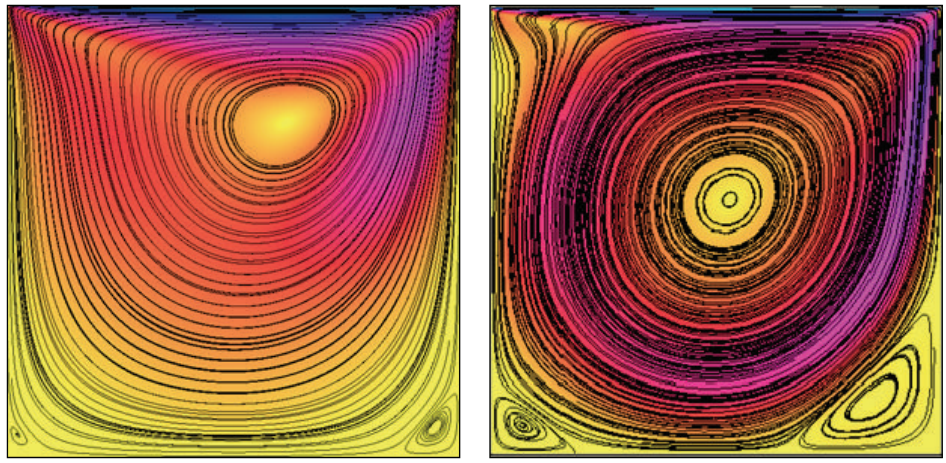
Parallel programs incur performance loss on cluster of computers due to load balancing problems caused by various reasons: irregular geometry, poor domain decomposition schemes, variable interprocessor communication latency, memory bandwidth limitations, and in general the nature of the parallel algorithm and differential equation being solved. Therefore, it is important to specify all the conditions under which scalability tests are performed. The first and most important factor is the hardware, compute units, and interconnect used for the simulation. In these days of multicore, multiprocessor, massively parallel heterogeneous clusters, a parallel CFD program can show significantly different scaling behaviors on different systems. The software, CFD solver, and message passing interface, used for the simulation, have the next big impact on performance. For instance, solvers with explicit time integration methods are known to scale very well on hundreds of thousands of cores. However, implicit methods are often used in CFD to avoid the CFL (Courant-Friedrichs-Lewy) limitation of explicit methods. The nature of the scalability test, that is, whether it is strong scalability for a fixed job or weak scalability for a fixed job per node, is another factor that causes confusion in reports of scalability tests.

Nonoverlapping domain decomposition methods usually scale very well on many cores, but they are not entirely embarrassingly parallel because information needs to be exchanged at interprocessor boundaries. Asynchronous iterations and communication-computation overlap with asynchronous communication are some of the methods used to hide communication latency. We measure the scalability of our parallel implementation for two cases: internode and intranode scalability. Intranode scalability tests are often used to demonstrate the memory bound nature of CFD programs; more cores do not necessarily translate to faster computation unless each core also gets enough bandwidth to keep it busy. The internode scalability is affected by intranode scalability by a constant factor [8]; therefore, it is preferable to test internode scalability with CPUs having few cores.

5.1. Internode Scalability. The system used to test the internode scalability is a cluster of 64 cores: 2 cores per node, AMD 1.6 GHz CPU, 2 GB RAM, fast Ethernet connection. We chose this system, with few number of cores per node, to avoid the effect of intranode scalability as mentioned previously. The fast Ethernet connection is a relatively slow interface compared to a modern InfiniBand interconnect. First, we simulate the lid-driven cavity problem with a grid size of 256 × 256 decomposed into a maximum of 36 subdomains. The turbulent Navier-Stokes equations are solved with a fully implicit time-stepping strategy. The speedup results for simulations with different number of cores are shown in Table 1. We can observe that speed up to 16 processors (8 nodes) is very good but from then onwards scaling suffers; this is most likely due to processors getting not enough work in this strong scaling test with a fixed size job. Therefore, we run the problem again with a larger grid size of 1024 × 1024 for a total of 1 million cells. As expected, this second run resulted in much better speedup numbers for the 25 and 36 processors case, as shown in Table 2, because of a higher computation to communication ratio. The 36 processors' case showed an improvement of 50% and the 25 processors' case a 16% increase. It was not possible to solve this bigger problem on one node due to insufficient memory; hence, the percentages are calculated relative to the 16 processors' result.

5.2. Intranode Scalability. The system used for measuring the intranode scalability is a 48-core AMD Opteron 6174 with 2.2 GHZ clock speed and 128 GB RAM. The lid-driven cavity problem is solved with a grid of 2048 × 2048 elements, and the speedup results are shown in Table 3. We can see that our program shows superlinear scalability up to 8 processors; however, the results with 16 and 32 processor are not as impressive. This is an indication that our CFD code is indeed memory-bound. Similar findings were reported for intranode scalability test of OpenFOAM CFD software [8].

5.3. Asynchronous versus Synchronous. The next system we use to test scalability and suitability of asynchronous



(a) Re = 100 (b) Re = 1000

(c) Re = 3200 (d) Re = 5000

Figure 5: Streamlines for different Reynolds numbers showing progressive formation of vortices at the bottom right corner → bottom left corner → top left corner.

Table 3: Intranode scalability for 2048 × 2048 case.

Processors	Time (ms)	Speedup
1	3094446	1.00
2	1483036	2.09
4	475617	6.5
8	372499	8.3
16	257947	12.0
32	219453	14.1

iterations has 512 cores: 64 cores per node, AMD Opteron 2.2 GHZ CPU, 256 GB RAM per node, and an InfiniBand interconnect. This cluster is used to measure the combined effect of intranode and internode scalability test because it is not suitable to measure either of them alone. Asynchronous Gauss-Seidel iterations are used to solve the lid-driven cavity problem with a grid of 2048 × 2048 elements. The results are shown in Table 4.

Table 4: Scalability for a 2048 × 2048 grid run with 512 processors.

Processors	Time (ms)	Speedup
1	2072764	1
2	1549708	1.33
4	648135	3.19
8	353380	5.86
16	144085	14.4
32	114001	18.2
64	74091	27.9
128	56386	36.7
256	44033	47.07
512	33115	62.59

The run time of computation using synchronous and asynchronous algorithms does not differ significantly for this problem, because of excellent load balancing in which each processor is working on exactly same size problem and

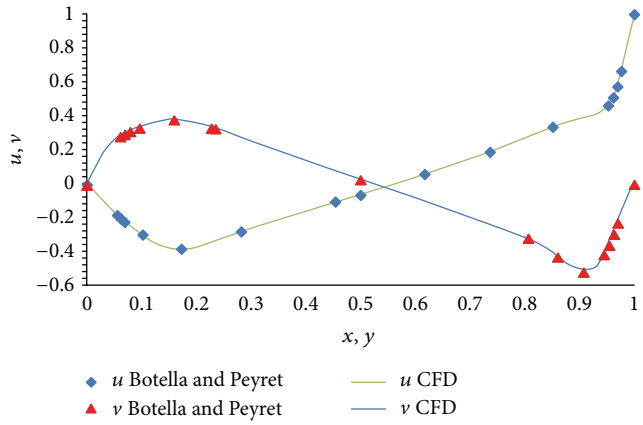
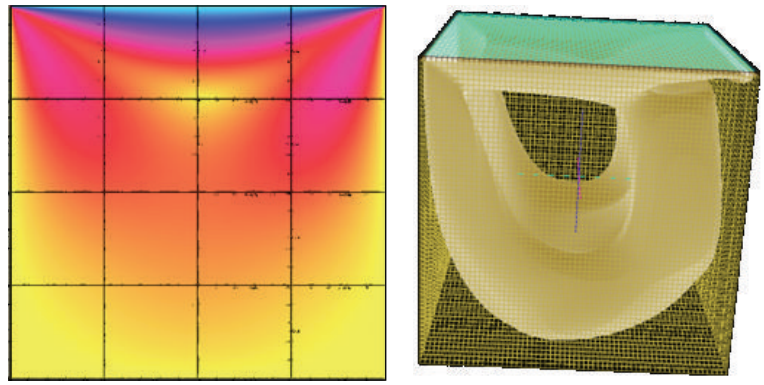


Figure 6: Horizontal (u) and vertical (v) velocity profiles along midvertical and midhorizontal sections at Re = 1000.



(a) (b)

Figure 7: Solution of 3D lid-driven cavity problem with 16 subdomains (a) and 3D isosurface plot showing flow pattern (b).

communicating through equal sized interfaces. Therefore, for this particular case, the synchronous implementation does not incur significant idle time to justify use of asynchronous scheme. On the other hand, if the subdomains were of unequal size, for example, one processor has twice larger subdomain size than the rest, or if one processor is twice slower than the rest, then the overall time of computation of the synchronous implementation would be governed by the speed of the slowest computational unit. The asynchronous implementation will try to cure this imbalance by doing more iterations on the faster unit. In general, for heterogeneous cluster with different computational units, asynchronous computation is preferable.

6. Conclusions and Recommendations

A CFD program is parallelized to run on a cluster of computers using the domain decomposition method and MPI for communication. The communication between subdomains can be done in a synchronous or asynchronous manner. Asynchronous communication is often used to reduce or hide completely communication latency by overlapping it with some kind of computation that can be done without waiting for the data to arrive. In this study, we examined performance of *asynchronous iterations* in the iterative solvers of a CFD program. Synchronous communication and iterations are usually the preferred choice in most industrial CFD code mainly because they are easier to implement and validate against serial version of the same program. However, asynchronous methods have a potential to alleviate some of the scalability problems faced on massively parallel systems, which are often due to collective communications at each iteration that force synchronization between participating processors. First, we validated the accuracy of our parallel implementation of the synchronous and asynchronous methods against the well-known lid-driven cavity benchmark problem. Once we are satisfied with the accuracy of the asynchronous implementation, we conducted scalability tests, both internode and intranode, on different systems. The asynchronous method was performed as well as the synchronous method, yielding a superlinear scaling for up to 8 cores and a speed up of about 23 on 36 cores. Both the synchronous and asynchronous methods did not perform well on the strong scaling test we conducted on the 512 cores machine we had available to us.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] D. S. Abdi and G. T. Bitsuamlak, "Wind flow simulations on idealized and real complex terrain using various turbulence models," *Advances in Engineering Software*, vol. 75, pp. 30–41, 2014.

[2] D. S. Abdi and G. T. Bitsuamlak, "Numerical evaluation of the effect of multiple roughness changes," *Wind and Structures*, vol. 19, no. 6, pp. 585–601, 2014.

[3] G. T. Bitsuamlak, T. Stathopoulos, and C. Bédard, "Numerical evaluation of wind flow over complex terrain: review," *Journal of Aerospace Engineering*, vol. 17, no. 4, pp. 135–145, 2004.

[4] H. Aboshosha, A. Elshaer, G. T. Bitsuamlak, and A. El Damatty, "Consistent inflow turbulence generator for LES evaluation of wind-induced responses for tall buildings," *Journal of Wind Engineering and Industrial Aerodynamics*, vol. 142, pp. 198–216, 2015.

[5] B. Blocken, T. Stathopoulos, and J. Carmeliet, "CFD simulation of the atmospheric boundary layer: wall function problems," *Atmospheric Environment*, vol. 41, no. 2, pp. 238–252, 2007.

[6] F. T. Johnson, E. N. Tinoco, and N. J. Yu, "Thirty years of development and application of CFD at Boeing Commercial Airplanes, Seattle," *Computers and Fluids*, vol. 34, no. 10, pp. 1115–1151, 2005.

[7] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI*, vol. 1, MIT Press, Cambridge, UK, 1999.

[8] M. Culpo, *Current Bottlenecks in the Scalability of OpenFOAM on Massively Parallel Clusters*, Partnership for Advanced Computing in Europe, 2010.

[9] R. Jia and B. Sundén, "Parallelization of a multi-blocked CFD code via three strategies for fluid flow and heat transfer analysis," *Computers and Fluids*, vol. 33, no. 1, pp. 57–80, 2004.

[10] W. Yong-Xian, Z. Li-Lun, L. Wei et al., "Efficient parallel implementation of large scale 3D structured grid CFD applications on the Tianhe-1A supercomputer," *Computers and Fluids*, vol. 80, no. 1, pp. 244–250, 2013.

[11] A. Povitsky and M. Wolfshtein, "Parallelization efficiency of CFD problems on a MIMD computer," *Computers & Fluids*, vol. 26, no. 4, pp. 359–371, 1997.

[12] P. K. Jimack and M. A. Walkley, "Asynchronous parallel solvers for linear systems arising in computational engineering," *Computational Technology Reviews*, vol. 3, pp. 1–20, 2005.

[13] M. Chau, P. Spiteri, R. Guivarch, and H. C. Boisson, "Parallel asynchronous iterations for the solution of a 3D continuous flow electrophoresis problem," *Computers and Fluids*, vol. 37, no. 9, pp. 1126–1137, 2008.

[14] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters," *Journal of Parallel Computing*, vol. 31, no. 5, pp. 439–461, 2005.

[15] N. Brown, J. Mark Bull, and I. Bethune, *Solving Large Sparse Linear Systems Using Asynchronous Multisplitting*, Partnership for Advanced Computing in Europe, 2013.

[16] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, Pa, USA, 2nd edition, 2003.

[17] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear Algebra and Its Applications*, vol. 2, no. 2, pp. 199–222, 1969.

[18] D. B. Szyld, "Perspectives on asynchronous computations for fluid flow problems," in *Computational Fluid and Solid Mechanics*, K. J. Bathe, Ed., pp. 377–380, Elsevier, 2001.

[19] A. Frommer and D. B. Szyld, "On asynchronous iterations," *Journal of Computational and Applied Mathematics*, vol. 123, no. 1-2, pp. 201–216, 2000.

[20] G. M. Baudet, "Asynchronous iterative methods for multiprocessors," *Journal of the ACM*, vol. 25, no. 2, pp. 226–244, 1978.

[21] E. M. Ortigosa, L. F. Romero, and J. I. Ramos, "Parallel scheduling of the PCG method for banded matrices rising from FDM/FEM," *Journal of Parallel and Distributed Computing*, vol. 63, no. 12, pp. 1243–1256, 2003.

[22] O. Botella and R. Peyret, "Benchmark spectral results on the lid-driven cavity flow," *Computers & Fluids*, vol. 27, no. 4, pp. 421–433, 1998.

[23] O. Goyon, "High-reynolds number solutions of Navier-Stokes equations using incremental unknowns," *Computer Methods in Applied Mechanics and Engineering*, vol. 130, no. 3-4, pp. 319–335, 1996.